Harvey Mudd College
Computer Science 42
Fall 2009

Assignment 4
User-Friendly Unicalc
Due. 11:59 p.m., Tue., 6 October 2009

This is the last part of a 3-part assignment, a command-line evaluator for Unicalc that is more user-friendly than the previous S-expression input version. This version will let the user enter expressions such as:

```
3 acre foot / hour + 2 liter / minute
```

returning answers that are Quantities, in the form:

```
1.0278791561991631 (meter meter meter) / second
```

This can be done by inserting a *parser* in front of the evaluator that you implemented in the previous assignment. You may also use my solution if yours has issues. We will use a $ symbol in front of variables so as not to confuse them with units. Here are some possible interactions with variables:

```
input > $x = 123
result: 123

input > $y = 456
result: 456

input > $x $y
result: 56088
```

The proper way to implement a parser is to first specify a *grammar* for the language to be parsed. I will give an appropriate grammar below, but I will not require that it be implemented completely. Floating-point input will be optional, for example.

Each input expression will be contained on a single line. A line is read into a string using (read-line). From there, it is suggested that you convert the string to a list and do the rest of the processing with lists. My specific recommendation is to adopt a uniform representation for the results of your parse functions. Each parse function is responsible for a different non-terminal in the grammar. The result of each would be a triple of the form

(*outcome result residual*)

where *outcome* is either **#t** or **#f** indicating whether parsing of the item was successful or not, *result* is the result of parsing, in the case of success, and *residual* is the residual unparsed input list. More specifically, result is an S expression that can be passed directly to your unicalc-eval function from assignment 3. In effect, the input to that function is an *abstract syntax tree* for the parsed expression.

I strongly recommend *not* attempting to parse full Unicalc Quantities, with numerator and denominator, from the input. Instead, parse numerals and units, and use your arithmetic functions to combine them into Quantities.

The grammar for expressions is given below. Words such as *input-line*, etc. are single non-terminals. Note that the symbols → | [ ] and ε are *meta* symbols, not part of the actual expression. Their meanings were discussed in class.

input-line → variable = expression

input-line → expression

expression → product rest-expression

rest-expression → ε | additive-operator product rest-expression

additive-operator → **+** | **-**

product → primary rest-product

rest-product → ε | [**/** | ε] primary rest-product

primary → numeral | unit | variable | (expression)

unit → letter rest-unit

variable → **$** rest-variable

rest-unit → ε | [letter | digit] rest-unit

rest-variable → ε | [letter | digit] rest-variable

numeral → digit rest-numeral

rest-numeral → ε | digit rest-numeral | **/** digit rest-fraction | **.** rest-fraction | e exponent

rest-fraction → ε | digit rest-fraction | e exponent

exponent → [**-** | ε] digit rest-exponent

rest-exponent → ε | digit rest-exponent

letter → a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
| _ | A | B | C | D | E | F | G | H | I | J | K | L
| M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9